COSTRUIAMO UN BACKEND SERVERLESS CON TYPESCRIPT, NODE.JS E AWS LAMBDA.

Amazon API Gateway | Aurora Serverless | AWS CloudFormation | AWS Lambda | GitHub | Lambda Custom Runtime | Nodejs | PostgreSQL | Serverless | Serverless Framework | TypeScript

beSharp | 21 Agosto 2020

Su **Amazon Web Services** il servizio computazionale Serverless per eccellenza rimane **AWS Lambda**, quasi immancabile in un'architettura che utilizza questo paradigma.

AWS Lambda permette di utilizzare potenza computazionale liberi dal pensiero di dover gestire i server sottostanti e senza doversi preoccupare di gestione di patch, update software o shutdown imprevisti della macchina. Utilizzando questo paradigma, possiamo concentrarci interamente sulla nostra applicazione.

AWS Lambda oggi offre numerosi **runtime engine**, oltre che la possibilità di creare il proprio se si vuole utilizzare un linguaggio di programmazione non ancora supportato in modalità managed da AWS.

Sta quindi a noi la scelta della tecnologia con cui scrivere le AWS Lambda Functions (FaaS): possiamo utilizzare **il linguaggio con cui siamo più familiari** o magari quello che ci permette di **raggiungere i nostri obiettivi più velocemente**.

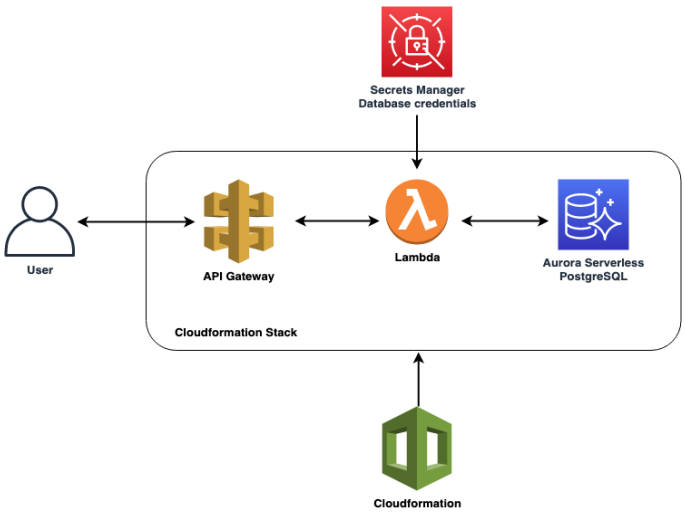
In questo articolo andremo a parlare di come sviluppare in modo veloce ed efficace **un backend serverless utilizzando TypeScript** come linguaggio di programmazione.

L'applicazione utilizzerà **Express**, un web application framework utile per lo sviluppo di servizi web per Node.js. **Node.js** non è altro che il runtime environment, già supportato da AWS, dove il nostro codice TypeScript, compilato in JavaScript, verrà eseguito su Lambda.

Andremo ad analizzare la soluzione proposta per poi deployarla sul nostro account AWS. Il progetto è consultabile e scaricabile da GitHub!

Servizi utilizzati

Iniziamo col proporre lo **schema architetturale** della nostra applicazione Serverless.



Il codice verrà rilasciato su una AWS Lambda function, la cui invocazione sarà possibile solamente attraverso un **API Gateway**.

In un applicativo backend non può ovviamente mancare un Data Source. Sfruttando a pieno il paradigma Serverless, andremo ad utilizzare un **Aurora Serverless** come database relazionale dove salvare e recuperare i nostri dati.

Ricapitolando, andremo ad utilizzare i seguenti servizi cloud:

- AWS Lambda
- API Gateway
- Aurora Serverless – Postgres Engine
- CloudFormation

Gestione dell'infrastruttura: il Serverless framework

Gli strumenti disponibili per la gestione di una infrastruttura di questo tipo sono numerosissimi: la CLI di AWS, la console o uno dei framework presenti oggi in rete come Troposphere, Terraform, o AWS SAM.

Come spiegato nell'introduzione, il nostro obiettivo è realizzare un'applicazione backend velocemente, per questo motivo **abbiamo scelto Serverless framework**!

Serverless è il framework scritto in Node.js che permette di gestire la lifecycle degli applicativi serverless. Ad oggi supporta numerosi Cloud Provider e funzionalità.

Per quanto riguarda AWS, Serverless ci permetterà di creare e gestire le risorse di cui abbiamo bisogno sul nostro account utilizzando **uno stack di Cloudformation**.

Prerequisiti

Prima di iniziare a lavorare sul progetto, è necessario aver soddisfatto i seguenti requisiti:

- Aver installato Node.js
- Aver installato Serverless framework
- Aver configurato correttamente la AWS CLI
- Aver un Aurora Serverless (Postgres engine) sul proprio account AWS

Siccome andremo a scrivere un'applicazione in **TypeScript**, Node.js sarà il runtime environment su cui andremo ad eseguire il nostro codice compilato in JavaScript.

Per poter deployare lo stack di Cloudformation su AWS, è necessario aver installato il Serverless framework sulla nostra macchina, tramite il comando *npm install -g serverless,* e aver configurato correttamente le credenziali della AWS CLI.

Hands on!

Ora che abbiamo dato le opportune premesse ed elencato i requisiti, possiamo **scaricare il progetto dal nostro repository GitHub** e vedere come è strutturato.

Iniziamo clonando il progetto:

git clone https://github.com/besharpsrl/blog-serverless-backend-nodejs

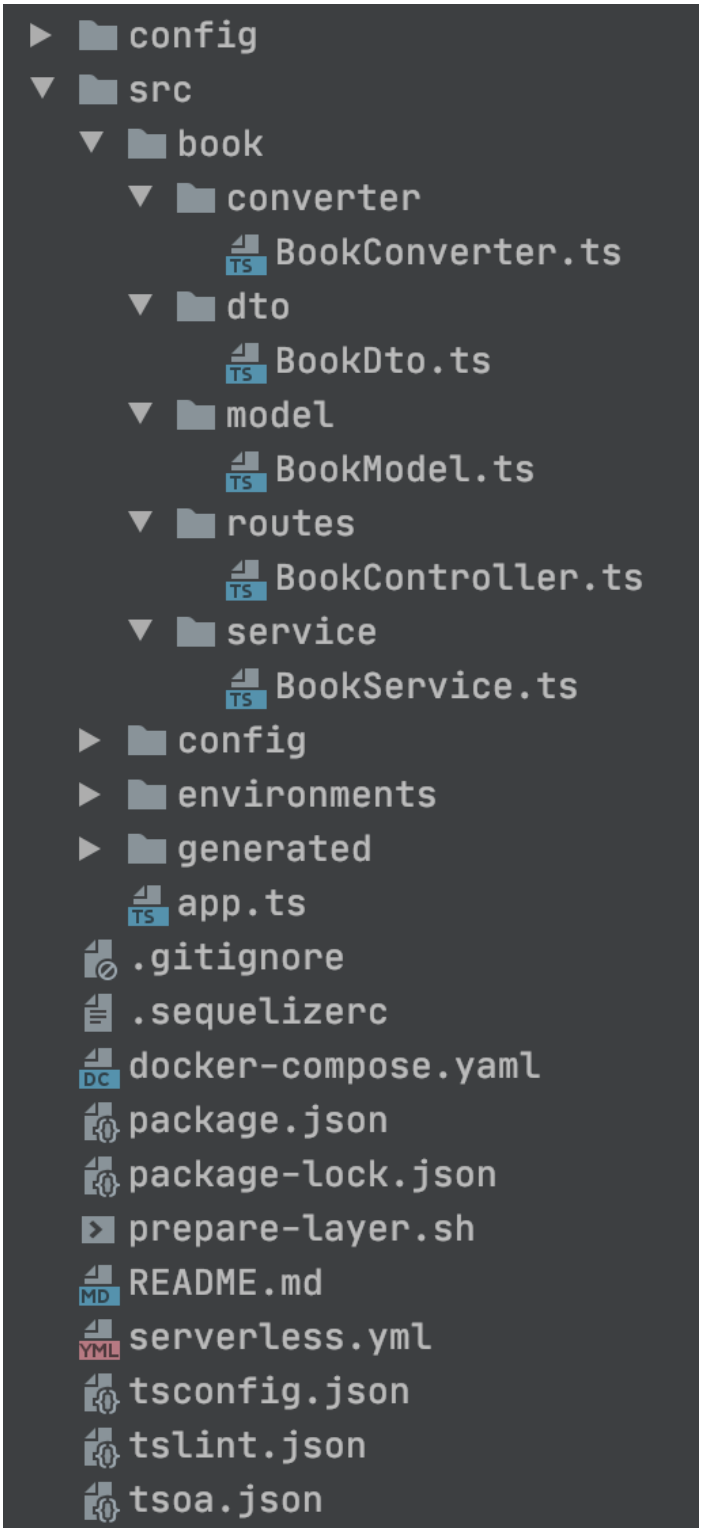
Successivamente non ci resterà che entrare nella cartella *blog-serverless-backend-nodejs* e lanciare il comando *npm install* per installare tutte le dipendenze necessarie.

Andiamo ad elencare quelle che meritano alcune precisazioni:

- **aws-sdk**: Software Development Kit per utilizzare i servizi di AWS. Nel nostro caso, lo utilizzerò per integrarci con Aurora Serverless e per recuperare le sue credenziali di accesso da Secrets Manager.
- **sequelize**: ORM molto utilizzato e supportato per Node.js.
- **serverless**: Framework per la creazione e gestione della infrastruttura Serverless.
- **express**: Web application framework minimale e flessibile per Node.js che offre una serie di funzionalità per lo sviluppo di applicazione web.
- **tsoa**: Framework utilizzato per scrivere i controller per l'autogenerazione delle rotte di Express. Grazie ad esso, è possibile, inoltre, generare OpenAPI specs valide sia per versione 2.0 che 3.0. Offre inoltre la gestione automatica per la validazione dell'input delle nostre request, senza dover creare e mantenere codice boilerplate.

Nel file *package.json* è possibile trovare queste e altre dipendenze di utility, oltre ai comandi che ci saranno utili per eseguire e testare le nostre API in locale e poi rilasciarle sull'account AWS.

Aprendo il progetto sul vostro IDE noterete uno scaffolding di questo tipo:



Entriamo ora nel dettaglio delle logiche applicative.

Descriveremo un caso d'uso molto semplice: delle semplici **API REST che offrono delle CRUD functions per gestire dei libri di una biblioteca.**

La logica risiede sotto la directory *book;* qui è definito il Controller di *tsoa*, il Service con la business logic e i modelli di *sequelize.*

Nel file *app.ts* andremo invece a registrare le rotte Express, autogenerate da tsoa a partire dai Controller, per poi esportare un modulo che verrà utilizzato come *handler* per la nostra Lambda function.

Il tutto è reso possibile dallo statement:

module.exports.handler = sls(app)

Con questo comando stiamo invocando **una funzione del Serverless framework** che andrà a creare un wrapper attorno alle rotte Express. L'handler della nostra AWS Lambda function saprà quindi a quale rotta Express dirottare le chiamate API REST.
Infine, non ci resta che analizzare il file *serverless.yml*:

```
service: express-serverless
provider:
  name: aws
  runtime: nodejs12.x
  region: eu-west-1
  stage: ${env:NODE_ENV}
  environment:
    NODE_ENV: ${env:NODE_ENV}
  iamRoleStatements:
    - Effect: 'Allow'
      Action:
        - 'secretsmanager:GetSecretValue'
      Resource:
        - '*'

package:
  exclude:
    - node_modules/**
    - defer" defer" defer" defer" defer" defer" defer" defer" defer" defer" src/**

layers:
  nodeModules:
    path: layer
    compatibleRuntimes:
      - nodejs12.x

…..

functions:
  app:
    handler: dist/app.handler
    layers:
      - {Ref: NodeModulesLambdaLayer}
    events:
      - http:
          path: /api
          method: ANY
          cors: true
      - http:
          path: /api/{proxy+}
          method: ANY
          cors: true
```

```
    vpc:
      securityGroupIds:
        - {Ref: LambdaSecurityGroup}
      subnetIds:
        - "subnet-1"
        - "subnet-2"
        - "subnet-3"

plugins:
  - serverless-offline
```

Proprio qui stiamo andando a definire le parti essenziali dell'infrastruttura, come il Cloud Provider su cui andremo a deployare la nostra Lambda function, il suo handler (il modulo che abbiamo esportato nel file *app.ts*) e i Lambda Layer da agganciare alla function.

Con questa configurazione, infatti, andremo a creare un Lambda Layer contenente tutti i node modules. Così facendo, alleggeriremo notevolmente le dimensioni della nostra funzione ottenendo vantaggi in **performance** durante la sua esecuzione.

Ma come verrà invocata la Lambda function? Tramite gli *events* di tipo *http* definiti nel file di Serverless*!* Questo permetterà la creazione di un API Gateway con cui sarà possibile richiamare la nostra funzione.

La risorsa col path */api/{proxy+}* sarà quella che farà da proxy alle rotte di backend. Abbiamo così creato un'unica risorsa lato API Gateway che ci permetterà di invocare tutte le REST API.

Local testing

Vi piacerebbe poter testare le API senza doverle necessariamente rilasciare su AWS Lambda ad ogni modifica? Il pacchetto di Node.js ***serverless-offline* è ciò che fa al caso nostro!**

Serverless-offline è un plugin di Serverless che emula AWS Lambda e API Gateway sulla nostra macchina per velocizzare le attività di sviluppo.

Prima di poterlo usare, però, sarà necessario creare un **database Postgres** in locale e lanciare le migrazioni di Sequelize:

Dalla root del progetto eseguiamo il comando

```
docker-compose up -d
```

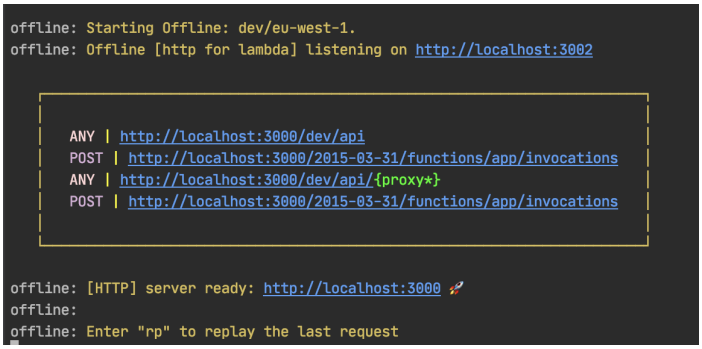
E poi:

```
npm run migrate-db-local
```

Ora che abbiamo il database in locale, non ci resta che provare le API! Lanciamo il comando *npm run start* per compilare il nostro codice TypeScript in JavaScript ed emulare la Lambda tramite il

plugin *serverless-offline*.

Non appena il comando avrà completato la sua esecuzione, otterremo il seguente output sul terminale:



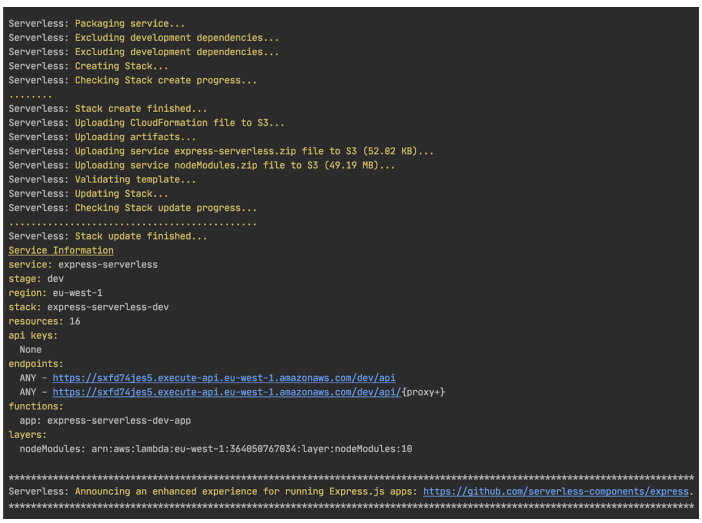
A questo punto siamo pronti per testare le API col nostro tool preferito (Postman, Curl). Utilizzando, ad esempio, il comando *curl* possiamo eseguire da terminale il comando:

curl http://localhost:3000/dev/api/book/

Deploy

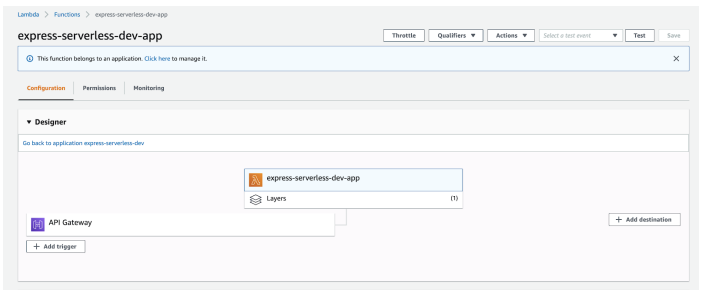
Ci siamo: è il momento di deployare l'infrastruttura sul nostro account!

Lanciando il comando *npm run deploy-dev* inizierà il processo di **rilascio**. La prima volta apparirà il seguente output:



Sull'account AWS potremo vedere che è stato generato uno stack Cloudformation contenente le risorse di cui abbiamo bisogno.

Accedendo alla console, tra la lista delle Lambda functions troveremo anche quella appena creata:



Prima di poter testare le API sarà necessario lanciare le migrazioni di *Sequelize* per creare le tabelle sul database Aurora Serverless.
Dobbiamo quindi poterci connettere al database su AWS. Creiamo una macchina bastion sull'account e dirottiamo il traffico dalla nostra macchina al bastion. Utilizziamo il comando *sshuttle:*

```
sshuttle --dns -r ubuntu@EC2_BASTION_IP YOUR_VPC_CIDR --ssh-cmd 'ssh -i YOUR_PEM_KEY'
```

A questo punto possiamo lanciare la migrazione tramite il comando:

```
npm run migrate-dev
```

Una volta completate le migrazioni su Aurora Serverless, proviamo le API attraverso API Gateway.

Nell'output del comando *npm run deploy-dev* è gia stato stampato l'endpoint dell'API Gateway, proviamolo subito:

curl https://sxfd74jes5.execute-api.eu-west-1.amazonaws.com/dev/api/book/

Il rilascio dell'infrastruttura Serverless sull'account AWS è finito!

Conclusioni

Per concludere, in questo articolo abbiamo visto come **creare un'applicazione Serverless su AWS Lambda utilizzando Node.js come runtime engine.**

Per scrivere il progetto abbiamo scelto **TypeScript** per avere il vantaggio di usare un linguaggio trascompilato, ampiamente supportato e conosciuto.

La scelta di Node.js come runtime engine ci ha permesso di avere un **Cold Start time molto contenuto** in quanto JavaScript viene direttamente interpretato dall'engine. Inoltre, grazie alla configurazione utilizzata, il codice sorgente è stato separato dalle dipendenze, salvate su Lambda Layer, abbattendo drasticamente le dimensioni del nostro sorgente e migliorando ulteriormente le performance di avvio.

Soddisfatti? 🙂

A presto su #Proud2beCloud per il prossimo articolo!

beSharp

Dal 2011 beSharp guida le aziende italiane sul Cloud. Dalla piccola impresa alla grande multinazionale, dal manifatturiero al terziario avanzato, aiutiamo le realtà più all'avanguardia a realizzare progetti innovativi in campo IT.

Get in touch

beSharp.it
proud2becloud@besharp.it